International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 4, July – August 2026
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

MANTIS: A Multi-Agent Architecture for Training and Benchmarking Neural Machine Translation Models

Jahnavi Somaraju[1], Nandyala Mamatha[2]
Assistant Professor[1], Final Year UG Student[2]
Department of Artificial Intelligence and Data Science, Aditya College of Engineering, Madanapalle
Email: mamathasree628166@gmail.com

Abstract:

Neural machine translation (NMT) research pipelines involve numerous loosely coupled stages — data curation, preprocessing, model training, hyperparameter search, evaluation, error diagnosis, and reporting — that are typically orchestrated manually or through rigid, script-based automation. This rigidity slows iteration, obscures provenance, and makes it difficult to reuse knowledge across experiments. We propose MANTIS, a multi-agent architecture in which specialized large language model (LLM) driven agents collaborate under a central orchestrator to plan, execute, evaluate, and report NMT training runs. MANTIS combines a supervisor agent for task decomposition and routing, a shared three-tier memory (short-term, long-term, and vector memory for retrieval-augmented generation), and a closed feedback loop that lets evaluation and error-analysis signals trigger automatic retraining or hyperparameter adjustment. We detail the agent roles, communication protocol, scheduling and conflict-resolution algorithms, and a reference implementation built on open frameworks. We describe an experimental protocol using WMT and FLORES-200 benchmark corpora (Costa-jussà et al., 2022), propose an evaluation methodology spanning translation quality, task completion rate, latency, cost, and scalability, and present an ablation design for isolating the contribution of each agent. The architecture targets a gap between single-agent LLM pipelines, which do not scale across concurrent experiments, and hand-orchestrated multi-tool pipelines, which do not adapt or reason about failures.

Keywords **— multi-agent systems; neural machine translation; large language model agents; retrieval-augmented generation; hyperparameter optimization; benchmarking**

I. INTRODUCTION

Neural machine translation (NMT) research is not a single computational step but a lifecycle: parallel corpora must be sourced and cleaned, text tokenized and subword-encoded, Transformer-based models trained and checkpointed (Vaswani et al., 2017), hyperparameters searched, translation quality benchmarked, failures diagnosed, and results reported before a system is ready for comparison or deployment. In most research groups this lifecycle is still driven either by hand, through notebooks and shell scripts stitched together by a researcher, or by monolithic automation that hard-codes a fixed sequence of steps. Neither approach adapts gracefully when a benchmark score regresses mid-experiment, a corpus turns out to be noisier than expected, or a new evaluation metric must be incorporated without restarting the pipeline from scratch.

Large language model (LLM) based agents offer a natural reformulation of this problem. Rather than a single script or a single generalist agent attempting every stage, each stage of the NMT lifecycle can be delegated to an agent with a narrow, well-defined responsibility, its own tools, and access to shared memory, while a coordinating agent manages dependencies between stages. Multi-agent decompositions of this kind have already shown gains over single-agent LLM pipelines in adjacent domains such as software engineering (Hong et al., 2024; Qian et al., 2024) and open-ended collaborative problem solving (Chen et al., 2024), motivating the same treatment for NMT experimentation.

This paper asks: given a translation task specification — a language pair, a domain, resource constraints, and evaluation criteria — how should a system of cooperating agents plan, execute, monitor, and iteratively improve an NMT training and benchmarking pipeline, while remaining auditable, recoverable from failure, and no more costly to run than manual experimentation or a single monolithic agent attempting every step itself? We argue that a multi-agent formulation is not merely convenient but necessary: separating data curation, training, search, and evaluation into distinct agents avoids conflating tools, prompts, and failure-handling logic in one unmanageable context; concurrent stages such as hyperparameter search and multi-language evaluation can proceed in parallel once a checkpoint is available; narrowly scoped agents are more consistent and checkable than a generalist agent juggling the entire pipeline; and a persistent, shared memory of prior experiments gives the system an institutional memory that a stateless, single-shot pipeline cannot have.

Building on this motivation, the paper makes four contributions. First, it proposes MANTIS, an agent-role taxonomy and orchestration design specific to the NMT training and benchmarking lifecycle, including a closed feedback loop that connects evaluation and error-analysis signals back into training. Second, it specifies a three-tier shared memory model — short-term, long-term, and vector memory — with a dedicated retrieval

agent that supports experiment reuse and retrieval-augmented configuration search. Third, it defines concrete algorithms for capability-based agent selection, dependency-aware task scheduling, and consensus-based conflict resolution between evaluation signals. Fourth, it defines an evaluation methodology and ablation protocol, spanning translation quality, task completion rate, latency, cost, scalability, and resource utilization, for benchmarking multi-agent NMT systems against single-agent and manual baselines.

II. Related Work

Multi-agent LLM systems. A growing body of work studies LLM agents as coordinated systems rather than isolated prompting techniques (Xi et al., 2023). AutoGen frames multi-agent collaboration as automated conversation among role-specialized agents (Wu et al., 2023); MetaGPT assigns software-engineering roles — product manager, architect, engineer — to distinct agents coordinated through structured output schemas (Hong et al., 2024); ChatDev similarly models software development as communication among role-specialized agents (Qian et al., 2024); and AgentVerse demonstrates that multi-agent groups can outperform single agents on reasoning and coding tasks, while also surfacing emergent behaviors such as conformity and free-riding that a coordinator must guard against (Chen et al., 2024). These frameworks establish that decomposing a complex task across specialized agents under a coordinating controller improves task success relative to a single long-context agent, and they converge on a small set of topologies: a star topology with a central orchestrator, a blackboard architecture in which agents read and write shared state, and peer-to-peer negotiation between agents with overlapping responsibilities. MANTIS adopts a hybrid of the first two, as described in Section IV.

Single-agent versus multi-agent pipelines. A single LLM agent prompted sequentially to plan an experiment, generate training configurations, and interpret results is simple to build and debug, but it concentrates all context and failure modes in one place: an early mistake, such as a malformed tokenizer configuration, propagates silently because the same agent cannot easily audit its own earlier decisions. Multi-agent pipelines trade this simplicity for parallelism, specialization, and mutual auditing, at the cost of coordination overhead, message-passing latency, and the engineering burden of shared state (Xi et al., 2023).

Limitations of existing systems for NMT. Existing NMT automation — fixed Fairseq or Marian training scripts, AutoML-style hyperparameter search — automates the search step but not the surrounding decisions, such as corpus selection, evaluation-metric choice, or error triage, that still require human judgement. Reported multi-agent frameworks, in turn, target general software or knowledge-work tasks rather than the specific lifecycle of NMT experimentation, in which training runs are long-running, checkpointed, GPU-bound processes rather than short tool calls, and few combine a retrieval-augmented memory of prior experiments with an explicit feedback loop that lets benchmarking results trigger new training runs automatically. Reported evaluations of multi-agent systems also rarely include ablations that isolate the contribution of individual agents, making it difficult to justify the added architectural complexity.

Research gap. No widely documented multi-agent architecture combines (a) coverage of the full NMT lifecycle from data curation to reporting, (b) a persistent, retrieval-augmented memory of prior experiments that informs new ones, (c) a closed loop between benchmarking results and subsequent training decisions, and (d) an ablation-based evaluation methodology that quantifies each agent's marginal contribution against single-agent and manual baselines. MANTIS is designed to close this gap.

III. Background

A multi-agent system (MAS) consists of multiple autonomous or semi-autonomous agents that perceive a shared environment, act on it through tools, and coordinate — directly or through a mediator — to achieve individual or collective goals. In LLM-based MAS, each agent is typically one or more LLM calls configured with a role-specific system prompt, a restricted tool set, and access to some portion of shared memory. LLMs act as agents, rather than pure text generators, when their output is used to select actions through tool or function calls, interpret the resulting observations, and decide when a (sub)task is complete (Yao et al., 2023); this requires structured prompting, a loop that feeds tool outputs back into the model's context, and explicit stopping criteria.

MANTIS assigns each agent a single primary responsibility and a narrow tool set, following the general finding that narrower scope improves both the reliability and the interpretability of agent behavior; Section IV defines each role in detail. Agents communicate through structured messages exchanged primarily via a central orchestrator, with a small number of direct peer channels reserved for tightly coupled agents where routing every message through the orchestrator would add unnecessary latency.

Memory in MANTIS is organized in three tiers. Short-term memory holds the live working context of the current task — recent messages, intermediate results, and the active plan — scoped to a single experiment session. Long-term memory is a relational store of completed experiments, configurations, metrics, and lineage, queryable by any agent for historical comparison. Vector memory holds dense embeddings of papers, past experiment summaries, and labeled error cases, retrieved by similarity to support retrieval-augmented generation (RAG) when an agent needs prior knowledge that will not fit in its context window (Lewis et al., 2020). Finally, planning in MANTIS is hierarchical: the orchestrator turns a high-level experiment request into an explicit directed acyclic graph (DAG) of subtasks with declared dependencies — evaluation depends on a checkpoint, which depends on preprocessing, which depends on data curation — while individual agents reason locally within their own subtask using a plan-act-observe loop.

IV. Proposed Architecture

MANTIS is organized as a star topology with a central Orchestrator/Supervisor agent, eight specialized worker agents, and a shared memory layer accessible to all agents (Fig. 1). A tool

layer beneath the agents exposes GPU clusters, translation frameworks — Hugging Face Transformers, Fairseq, Marian — and MT metric libraries as callable tools. The orchestrator itself holds no domain-specific NMT knowledge; its responsibilities are purely coordinative: decomposing an experiment request into subtasks, selecting which agent should handle each subtask, enforcing dependency order, propagating budget and time constraints, and mediating disagreements between agents (Section V-D).

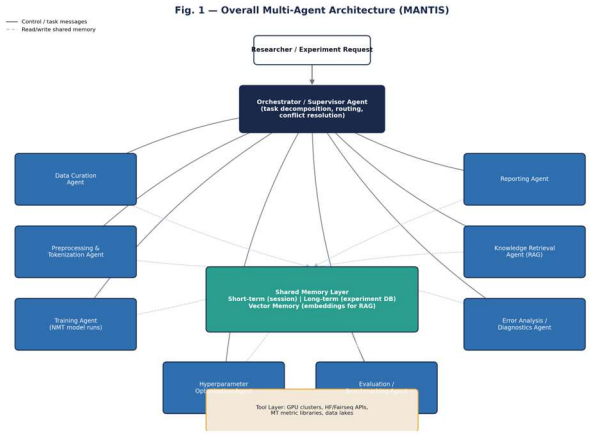

Fig. 1. Overall multi-agent architecture. The orchestrator routes tasks to worker agents; all agents read and write a shared memory layer beneath them.

A. Agent Roles

Table I summarizes the nine agents and their primary responsibilities. Each is scoped to a single stage of the NMT lifecycle and given only the tools that stage requires, so that no agent can take an action outside its declared role.

TABLE I AGENT ROLES AND RESPONSIBILITIES IN MANTIS

Agent	Primary responsibility
Orchestrator / Supervisor	Decomposes the experiment request into a task DAG, routes subtasks, resolves conflicts, tracks overall progress and budget.
Data Curation Agent	Sources, filters, and deduplicates parallel corpora; flags domain mismatch and quality issues; produces a corpus manifest.
Preprocessing Agent	Cleans, normalizes, tokenizes, and subword-encodes (BPE / SentencePiece) text; builds vocabulary and train/dev/test splits.
Training Agent	Launches and monitors NMT training runs, manages checkpoints, reports intermediate validation metrics.
Hyperparameter Optimization (HPO) Agent	Proposes and refines hyperparameter configurations (learning rate, batch size, architecture width/depth) using search informed by past runs in long-term memory.
Evaluation / Benchmarking Agent	Runs the benchmark suite (BLEU, chrF++, COMET, TER) on held-out and standard test sets; assigns confidence to each score.
Error Analysis / Diagnostics Agent	Inspects translation outputs for systematic failure patterns — hallucination, omission, mistranslation of named entities — and proposes root causes.
Knowledge Retrieval Agent (RAG)	Embeds queries from other agents, retrieves relevant prior experiments, papers, and known error cases from vector memory, and injects retrieved context into requesting agents' prompts.
Reporting Agent	Aggregates results, ablations, and plots into a structured report and dashboard for the researcher.

B. Workflow and Coordination

A typical cycle proceeds as follows. The researcher submits an experiment specification; the orchestrator builds a task DAG and dispatches the first ready nodes; Data Curation and Preprocessing prepare the corpus; Training launches with an initial configuration while the HPO agent proposes refinements between runs; Evaluation scores each checkpoint of interest; Error Analysis inspects low-scoring outputs and may request a further training cycle through the orchestrator, closing the feedback loop; and Reporting compiles the final artifact. Fig. 2 renders this cycle as a sequence diagram of the messages exchanged between agents, and Fig. 3 renders the same cycle as an end-to-end workflow with the feedback loop made explicit.

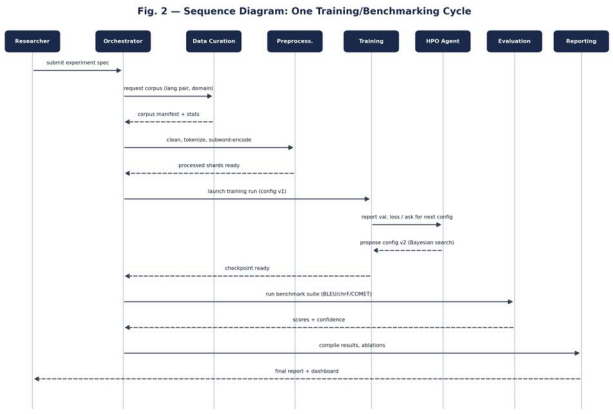

Fig. 2. Sequence diagram of one training/benchmarking cycle, showing message flow between the researcher and each agent.

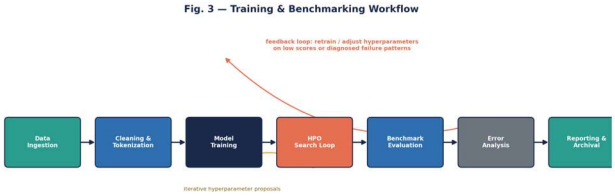

Fig. 3. End-to-end workflow with the retraining/adjustment feedback loop triggered by evaluation and error-analysis signals.

C. Task Decomposition and Routing

On receiving an experiment specification, the orchestrator produces a DAG whose nodes are subtasks — curate corpus, tokenize, train (v1), evaluate (v1), diagnose (v1), train (v2), and so on — and whose edges encode data dependencies. Decomposition follows a fixed template for the standard NMT lifecycle,

augmented with additional nodes when the request specifies non-default steps such as domain adaptation or low-resource fine-tuning. Routing is capability-based: each worker agent publishes a capability descriptor, an embedding of its role description together with its tool list, and when a DAG node becomes ready the orchestrator embeds the node's task description and routes it to the agent whose capability descriptor is closest by cosine similarity, subject to a minimum-similarity threshold (Section V-A). Below this threshold the orchestrator flags the task as ambiguous and requests clarification rather than guessing.

D. Shared Memory and Retrieval

All agents read and write the three-tier memory introduced in Section III (Fig. 4). Short-term memory is scoped per experiment session and discarded on completion; long-term memory persists indefinitely in a relational store; vector memory persists indefinitely in a vector database and grows as further experiments and papers are ingested. The Knowledge Retrieval agent is the sole writer of new vector-memory entries and the primary reader on behalf of other agents: when, for example, the HPO agent needs to know which learning rates worked for a similar language pair, it issues a retrieval request rather than querying the vector store directly, which lets the Knowledge Retrieval agent apply consistent chunking, filtering, and recency weighting across all consumers (Lewis et al., 2020).

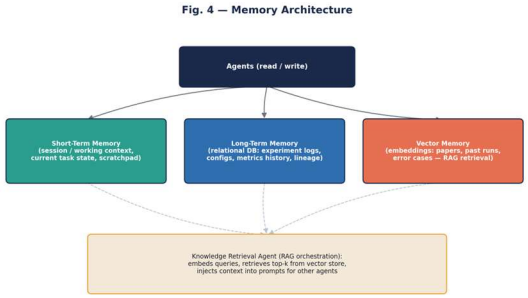

Fig. 4. Memory architecture: agents read/write short-term, long-term, and vector memory; the Knowledge Retrieval Agent mediates RAG access to vector memory.

E. Feedback Loop and Error Handling

Evaluation and Error Analysis outputs are not terminal. If a benchmark score falls below a target threshold, or Error Analysis identifies a correctable pattern such as a truncated vocabulary causing rare-word failures, the orchestrator opens a new Training subtask with an adjusted configuration, closing the loop shown in Fig. 3; a maximum number of feedback iterations is configured per experiment to bound cost. Robustness is addressed at several levels: tool-call failures such as a training-job crash are retried with exponential backoff before being escalated to the orchestrator, which may reassign the subtask or request human intervention; every inter-agent message is schema-validated so that malformed payloads cannot propagate downstream; long-running subtasks are monitored by heartbeat so that a stalled job is flagged rather than silently treated as complete; and evaluation results that conflict with Error Analysis conclusions invoke the conflict-resolution procedure of Section V-C rather than one output silently overriding the other.

V. ALGORITHMS

This section formalizes the four algorithmic procedures that govern coordination in MANTIS: agent selection, task scheduling, conflict resolution, and consensus. Each is presented as pseudocode together with the reasoning behind its design.

A. Agent Selection

Capability-based routing selects the agent a* for a task t by maximizing the cosine similarity between the task embedding and each candidate agent's capability embedding, subject to a minimum-similarity threshold τ. If no candidate clears τ, the task is marked ambiguous rather than dispatched to the closest-but-poor match, and ties within a small epsilon band are broken by current agent load, giving basic load balancing across parallel HPO trial instances.

Algorithm 1. Capability-Based Agent Selection

```
Input: task t, candidate agents A = {a1,...,an}, threshold τ, tie band ε
e_t ← embed(description(t))
for each a_i in A: s_i ← cos(e_t, capability(a_i))
a* ← argmax_i s_i
if s_{a*} < τ then
    return AMBIGUOUS   // request human clarification
C ← { a_i : s_i ≥ s_{a*} - ε }   // near-ties
if |C| > 1 then a* ← least-loaded agent in C
return a*
```

B. Task Scheduling

Subtasks are drawn from the orchestrator's DAG through a dependency-aware priority queue. A node becomes eligible once every predecessor is complete; among eligible nodes, priority favors the critical path — the node with the longest remaining dependency chain — then researcher-declared priority, then first-in-first-out order. This minimizes end-to-end wall-clock time by never idling the critical path while a lower-priority branch executes.

Algorithm 2. Dependency-Aware Scheduling

```
Input: DAG G = (V, E) of subtasks with declared priorities
Ready ← { v in V : all predecessors of v are complete }
while Ready is not empty:
    v* ← node in Ready maximizing (critical_path_length(v),
          priority(v), -arrival_time(v))
    dispatch(v*) to agent selected by Algorithm 1
    remove v* from Ready; on completion, add newly-eligible
```



```
        successors to Ready
```

C. Conflict Resolution

Conflicts arise chiefly between Evaluation and Error Analysis — for example, Evaluation reports an acceptable BLEU score while Error Analysis flags a systematic entity-translation failure — or between repeated Evaluation runs with divergent scores due to non-determinism. The orchestrator first re-runs the disputed measurement once if this is cheap; if disagreement persists, it weights each agent's claim by self-reported confidence and by historical reliability tracked in long-term memory to compute a resolved verdict; and if the weighted verdict remains within a small margin of the decision boundary, the conflict is surfaced to the researcher rather than resolved silently.

Algorithm 3. Conflict Resolution Between Agent Signals

```
1: Input: conflicting claims c1, c2 from agents
a1, a2, margin m
2: if cheap_to_rerun(c1, c2) then rerun once and
update c1, c2
3: if c1 and c2 still disagree:
4:    w1 ← confidence(c1) × reliability(a1);
      w2 ← confidence(c2) × reliability(a2)
5:    verdict ← (w1·c1 + w2·c2) / (w1 + w2)
6:    if |verdict - decision_boundary| < m then
7:      escalate_to_researcher(c1, c2, verdict)
8:    else return verdict
```

D. Consensus Mechanism

Decisions that depend on more than two contributing signals — for instance, accepting a checkpoint as the new best model, which depends jointly on Evaluation, Error Analysis, and cost/latency from the Training agent — use a weighted-majority consensus. Each contributing agent casts a normalized vote in [-1, 1]; votes are weighted by role-specific priors, with Evaluation weighted highest for quality decisions and Training weighted highest for feasibility decisions; and the checkpoint is accepted only if the weighted sum exceeds a configured acceptance margin.

Algorithm 4. Weighted-Majority Consensus

```
1: Input: votes v1..vk in [-1,1] from agents
a1..ak,
   role weights w1..wk, margin μ
2: score ← Σ_i (w_i · v_i) / Σ_i w_i
3: if score > μ then return ACCEPT
4: else if score < -μ then return REJECT
5: else return ESCALATE  // ambiguous, defer to
researcher
```

VI. Implementation

The reference implementation composes existing, reproducible tooling rather than bespoke infrastructure. Orchestration is implemented on a graph-based agent framework in the style of LangGraph or AutoGen (Wu et al., 2023), which supplies the DAG scheduler and message bus described in Section V. NMT-specific tooling is provided by Hugging Face Transformers and Fairseq/Marian for model definitions and training loops, and sacreBLEU (Post, 2018) together with COMET (Rei et al., 2020) for evaluation. Each agent role is backed by an instruction-following LLM exposed through a standard chat-completions API with function-calling enabled, so that agent reasoning and tool invocation share one interface across roles.

Persistent state is split across three stores matching the memory tiers of Section III: PostgreSQL holds long-term relational memory — experiments, configurations, metrics, and lineage — with foreign keys forming a queryable lineage graph (e.g., "show every run for language pair X ordered by COMET score," or "replay the message trace for run Y"); a vector database (such as Chroma, Weaviate, or pgvector) indexes chunked paper abstracts, past run summaries, and labeled error cases, each tagged with language-pair and domain metadata for retrieval filtering; and Redis provides short-term session memory and the message queue. Training and hyperparameter-search jobs run as containerized GPU workloads scheduled through Kubernetes Jobs or a workload manager such as Slurm, behind a thin job-launcher API that the Training agent calls rather than managing compute directly.

Each worker agent is deployed as an independent service exposing a single tool-callable endpoint for accepting tasks and a status endpoint for health checks; the orchestrator is the only component permitted to dispatch tasks directly, aside from the sanctioned Training–HPO peer channel described in Section III. The orchestrator, memory services, and reporting dashboard run as always-on services, while Training and HPO agent instances scale elastically with the GPU job queue — Fig. 1 doubles as a logical deployment view once each labeled box is read as a containerized service rather than a purely logical role.

VII. Experimental Setup

We propose evaluating MANTIS on a combination of high-resource and low-resource translation settings. WMT14 English–German and WMT19 English–Russian provide high-resource bilingual benchmarks with well-established prior results; FLORES-200 (Costa-jussà et al., 2022) provides multilingual, low-resource language pairs that stress-test the Data Curation and Error Analysis agents on sparse corpora; and a held-out domain-specific corpus, such as a biomedical or legal collection, introduces a deliberate distribution shift to test whether the feedback loop of Section IV-E detects and corrects degraded performance.

Four scenarios follow from these corpora: high-resource bilingual training and benchmarking on a single, data-rich language pair; low-resource or multilingual fine-tuning, which exercises RAG-based configuration retrieval from similar prior experiments; domain adaptation under an injected distribution shift, to evaluate whether the feedback loop detects and corrects degraded performance rather than merely reporting it; and concurrent multi-experiment load, to evaluate scheduling and scalability under contention for shared GPU resources (Section V-B).

Each scenario is run under three conditions: a manual baseline, in which a human researcher orchestrates the same underlying training and evaluation tools through scripts; a single-agent baseline, in which one LLM agent is given the same tools and asked to perform every stage sequentially in one long-running

session without role specialization or a supervisor; and MANTIS (full), the proposed architecture with all nine agents and the closed feedback loop active. All three conditions share the same corpus, compute budget, and target metrics, and are repeated across multiple seeds and trials to account for training and search stochasticity. Section VIII defines the metrics collected across these conditions; Section IX reports the intended comparative and ablation analysis.

VIII. Evaluation Metrics

Two complementary families of metrics are collected: translation-quality metrics, which assess the NMT models produced, and system-level metrics, which assess the multi-agent pipeline itself rather than its output. For translation quality we report BLEU (Papineni et al., 2002) and chrF++ (Popović, 2015) for surface-level n-gram overlap with references, COMET (Rei et al., 2020) for learned, semantically-aware quality estimation, and TER (Snover et al., 2006) as a complementary edit-based metric; together these span the lexical-overlap and neural-metric families that WMT shared tasks treat as complementary rather than substitutable.

For system-level behavior we report output quality, human- or LLM-judged adequacy and fluency ratings on a sampled subset of translations; task completion rate, the fraction of submitted experiment specifications that reach a final report without an unresolved error; latency, the wall-clock time from request submission to final report together with a per-stage breakdown; cost, the total compute cost in GPU-hours and the LLM API cost in token spend per completed experiment; scalability, the degradation in completion rate and latency as the number of concurrent experiments increases; and resource utilization, GPU and CPU utilization during training and search phases together with idle time attributable to scheduling overhead. Reporting both families together is what lets Section IX's ablation distinguish an agent that improves translation quality from one that merely improves system efficiency.

IX. Results

The tables below specify the intended reporting format for a completed MANTIS evaluation rather than reporting results from a deployment already run; cells marked [fill] are placeholders for measurements to be collected under the protocol of Section VII. We include them at this level of detail because a central claim of the paper — that the architecture's added coordination is justified by measurable gains — is only as credible as the comparative and ablation evidence that would populate these tables.

A. Comparative Analysis

Table II compares the manual baseline, the single-agent baseline, and MANTIS (full) on the high-resource scenario of Section VII, reporting translation quality alongside the system-level metrics of Section VIII. We would expect MANTIS to trade a latency and cost premium, relative to the manual baseline, for higher task completion and translation quality relative to the single-agent baseline, whose lack of specialization and mutual auditing should produce more silent failures under the same budget.

TABLE II Comparative Results Across Baselines And MANTIS (High-Resource Scenario)

Condition	COMET ↑	BLEU ↑	Completion %	Latency (h)	Cost (USD)
Manual baseline	[fill]	[fill]	[fill]	[fill]	[fill]
Single-agent baseline	[fill]	[fill]	[fill]	[fill]	[fill]
MANTIS (full)	[fill]	[fill]	[fill]	[fill]	[fill]

Report mean and standard deviation over seeds.

B. Ablation Study

The ablation removes one agent or module at a time from the full MANTIS configuration, substituting a no-op or a simplified rule-based stand-in, and re-runs the same scenario, isolating each component's marginal contribution to quality and system metrics (Table III). Larger gaps relative to the full configuration indicate a more critical component; we would expect removing the feedback loop to most affect COMET on the domain-adaptation scenario specifically, since that scenario is designed to require a corrective retraining cycle, while removing the HPO agent should mainly affect latency and cost rather than final quality if the fixed default configuration is already reasonable.

TABLE III Ablation Study Template

Configuration	COMET ↑	Completion %	Latency (h)
Full MANTIS	[fill]	[fill]	[fill]
Minus HPO agent (fixed hyperparameters)	[fill]	[fill]	[fill]
Minus Error Analysis agent (no diagnosis)	[fill]	[fill]	[fill]
Minus Knowledge Retrieval / RAG	[fill]	[fill]	[fill]
Minus feedback loop (single training pass)	[fill]	[fill]	[fill]

Larger gaps versus the full configuration indicate a more critical component.

C. Scalability

A scalability sweep should report task completion rate and mean latency as the number of concurrent experiment requests increases — for example 1, 4, 8, and 16 — for both the single-agent baseline and MANTIS, to test whether the parallelism advantage argued for in Section I holds under contention for shared GPU resources rather than only in the uncontended case.

X. Discussion

MANTIS's modularity is its principal strength: individual agents, such as the evaluation metric suite, can be upgraded without redesigning the surrounding pipeline, and the closed

feedback loop automates a decision — whether and how to retrain — that is otherwise manual and easy to defer indefinitely. Persistent, retrieval-augmented memory further lets later experiments benefit from earlier ones in a way that neither the manual nor the single-agent baseline does systematically, since neither maintains a queryable record of what configurations and failure patterns have already been observed.

These strengths come with corresponding limitations. Coordination overhead — message passing, routing, and consensus computation — adds latency and cost relative to a well-tuned manual script for language pairs that are already well understood, so the architecture's advantage is expected to grow with task novelty and concurrency rather than being uniform across all settings. The agent-selection and conflict-resolution thresholds introduced in Section V require calibration per deployment and may not transfer across substantially different compute budgets. Evaluation and error-analysis agents also inherit any biases or blind spots of the underlying LLM judge used for output-quality assessment, so system-level metrics should be read alongside, not instead of, the translation-quality metrics of Section VIII.

Several failure modes are worth anticipating explicitly. Ambiguous task descriptions that fall below the routing similarity threshold τ stall until a human clarifies, which can become a bottleneck if clarification requests are frequent; conflicting evaluation signals near the consensus decision boundary of Section V-D require human adjudication, adding latency in precisely the cases that are hardest to resolve automatically; and subtle corpus-quality issues — a systematic but low-frequency mistranslation pattern in the training data, for instance — may not be caught by Data Curation and can propagate until Error Analysis observes their downstream effect.

Security and privacy considerations follow from the same shared-memory design that gives MANTIS its institutional memory. Corpora and model outputs may contain sensitive or personal text, so access to long-term and vector memory should be scoped and audited, particularly for the Knowledge Retrieval agent, which aggregates across experiments and is consequently exposed to more data than any single worker agent. Restricting each agent's tool set (Section IV) limits the blast radius if an agent is compromised or misbehaves, but the orchestrator's broad routing privileges make it the highest-value target for hardening. Finally, persisting every inter-agent message for auditability (Section V-C) means that log itself must be access-controlled, since it can reveal proprietary corpora, configurations, and unpublished results.

XI. Future Work

Four extensions follow naturally from the design presented here. Self-learning agents could update their own prompts or routing embeddings from historical success rates rather than relying on static configuration, closing a second-order feedback loop around the one described in Section IV-E. Dynamic agent creation would let the orchestrator instantiate a new specialized agent — a script-specific tokenization expert, for instance — on demand for an unusual language pair, rather than requiring every possible specialization to be anticipated at design time. Adaptive planning would let the task DAG be revised mid-execution in response to intermediate results, rather than following the fixed decomposition template of Section IV-C. And richer human-in-the-loop collaboration would let a researcher intervene at any DAG node, override a consensus decision, or supply guidance that is persisted into long-term memory for future experiments, turning isolated human corrections into durable institutional knowledge.

XII. Conclusion

This paper proposed MANTIS, a multi-agent architecture for training and benchmarking NMT models that decomposes the experimentation lifecycle across nine specialized agents coordinated by a central orchestrator, backed by a three-tier shared memory and a closed feedback loop between evaluation, error analysis, and retraining. We detailed the communication protocol, routing, scheduling, conflict-resolution, and consensus algorithms that coordinate these agents; a reference implementation built on open frameworks; and an experimental and evaluation methodology, including an ablation design, intended to quantify each agent's contribution against manual and single-agent baselines. The architecture targets a gap between rigid, hand-orchestrated NMT pipelines and single-agent LLM automation that does not scale or reason well across the full lifecycle. Beyond NMT, the same structure — long-running, checkpointed training; iterative hyperparameter search; and benchmark-driven feedback — recurs in adjacent domains such as speech recognition and multimodal model development, suggesting MANTIS's coordination design may generalize beyond the setting studied here.

References

[1] Chen, W., Su, Y., Zuo, J., Yang, C., Yuan, C., Chan, C., Yu, H., Lu, Y., Hung, Y.-H., Qian, C., Qin, Y., Cong, X., Xie, R., Liu, Z., Sun, M., & Zhou, J. (2024). AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In Proc. ICLR.

[2] Costa-jussà, M. R., Cross, J., Çelebi, O., Elbayad, M., Heafield, K., Heffernan, K., et al. (NLLB Team). (2022). No Language Left Behind: Scaling human-centered machine translation. arXiv:2207.04672.

[3] Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., & Schmidhuber, J. (2024). MetaGPT: Meta programming for a multi-agent collaborative framework. In Proc. ICLR.

[4] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In Proc. NeurIPS.

[5] Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). BLEU: A method for automatic evaluation of machine translation. In Proc. 40th Annual Meeting of the ACL.

[6] Popović, M. (2015). chrF: character n-gram F-score for automatic MT evaluation. In Proc. Tenth Workshop on Statistical Machine Translation (WMT).

[7] Post, M. (2018). A call for clarity in reporting BLEU scores. In Proc. Third Conference on Machine Translation (WMT).

[8] Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., & Sun, M. (2024). ChatDev: Communicative agents for software development. In Proc. 62nd Annual Meeting of the ACL.

[9] Rei, R., Stewart, C., Farinha, A. C., & Lavie, A. (2020). COMET: A neural framework for MT evaluation. In Proc. EMNLP.

[10] Snover, M., Dorr, B., Schwartz, R., Micciulla, L., & Makhoul, J. (2006). A study of translation edit rate with targeted human annotation. In Proc. 7th Conference of the AMTA.

[11] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In Proc. NeurIPS.

[12] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., & Wang, C. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155.

[13] Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., et al. (2023). The rise and potential of large language model based agents: A survey. arXiv:2309.07864.

[14] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. In Proc. ICLR.

